Sept. 5-9, 2004

Maths for Biomedical Engineering

University of Warwick

Daniel Kaplan, Macalester College

Activities for Nonlinear Dynamics Signal Processing

1 Linearity and Nonlinearity

This set of exercises is about the reasons why nonlinearity can be important in interpreting biomedical signals. We will use both linear and simple nonlinear techniques to study some model signals and see how rather simple nonlinear mechanisms can lead to behavior that is profoundly different from that of linear systems. Later in other exercises, we will use more sophisticated nonlinear techniques to try to capture important aspects of the nonlinear mechanisms through signal analysis.

Setting Up

1. To start, make sure that you have access to the software and data sets that we will use throughout these exercises. All of the software is written in MATLAB. The software m-files and the data files are contained in a directory that will be announced. Add this directory, and the subdirectories, to the MATLAB path.

   You can test whether this has been done by giving the simple MATLAB command,
   ≫ `help lagembed`
   If an error message appears, then you do not have the MATLAB path properly set.

2. Make sure that you are comfortable writing MATLAB programs or scripts, saving the m-files in an appropriate location, and so on. As an exercise that we will use later today, write a MATLAB function that takes calculates an FFT estimate of the power spectrum of a signal. Your function should take two arguments: the signal (stored as a vector) and the sampling frequency (stored as a single number). It should return two values: a vector containing the power spectrum and a vector of exactly the same length containing the frequency (in Hz) of each point in the power spectrum. You can make your program as fancy or as simple as you like: perhaps you will want to have a simple periodogram, perhaps a windowed periodogram, etc.

The Quadratic Map

One of the most famous nonlinear dynamical systems is the *Quadratic Map*, a system whose dynamics are described by a single, one-variable, finite-difference equation:

$$x_{t+1} = \mu x_t (1 - x_t).$$

The *state variable* x_t is a number between 0 and 1. The parameter μ is typically taken to be between 0 and 4.

As the parameter μ changes, the behavior of the system changes. For $\mu < 3$, there is a stable fixed point and the time series approaches a stable equilibrium. As μ increases from 3 to 4, the system undergoes a sequence of period-doubling bifurcations and chaos emerges.

$3.0000 < \mu < 3.4495$	stable cycle of period 2
$3.4495 < \mu < 3.5441$	stable cycle of period 4
$3.5441 < \mu < 3.5644$	stable cycle of period 8
$3.5644 < \mu < 3.5688$	stable cycle of period 16
$3.5688 < \mu < 3.5700$	longer periods of length 2^n
$3.5700 < \mu < 4.0000$	aperiodic behavior (chaos!) as well as narrow ranges with periodic behavior

Real biomedical signals are not as simple as the one from the quadratic map, but we can think of the quadratic map as a metaphor. In particular, we can imagine that the parameter μ might be some physiological parameter that we want to monitor. The monitoring question is: By analyzing the time series, can we tell that a parameter has changed? Or, given two time series, can we tell if they are at a different parameter from one another?

We are going to explore the behavior of this dynamical system using several tools:

- Time series plots. Just looking at the state x_t plotted against t can be helpful.
- Density plots. A histogram of the state variable. (The MATLAB program `hist` can be used for this.
- The power spectrum.
- Finding the period.

Your task is:

1. Write a program that iterates the quadratic map. This program should take 3 arguments: a value of μ, the number of iterations N to carry out, an initial value x_0 between 0 and 1. The program should return the time series containing the iterates x_0 to x_N.

2. Try values of mu in each of the behavior regimes listed in the above table.

- How does the behavior described in the table appear in the plot of x_t versus t?

- How does the behavior appear in the histogram?

- How does the behavior appear in the power spectrum? Hint: a period-2 behavior will give a spike at the Nyquist frequency. A period-4 behavior will, in theory, have a spike at half the Nyquist frequency. But, be careful! Depending on the length of the segments that you analyze compared to the period of the signal, those spikes might be spread out over a range of frequencies.

  Pay particular attention to the amplitude of the spikes. Are the high-period spikes similar in height to the Nyquist frequency spike?

- The program `distVSlag` will help to estimate the period of the signal in the time domain. It does this by calculating how far the state x_t is from the final state x_N for different lags. When the distance is zero, the system has returned exactly to a previous state and so is periodic with that lag. Of course, noise and transients may prevent the system from returning exactly to a previous state. Use it like this
  ≫ `[lags, dists] = distVSlag(sig);`

  ≫ `plot(lags, dists)`
  The smallest lag for which the distance is approximately zero is the (approximate) period of the system.

Another questions to think about: How do you think that measurement noise would affect the different ways of judging the parameter from the time series?

The Sleep-Wake Cycle

The quadratic map is not a model of any particular physiological process, but there are simple models that are. We are going to work with one called the Poincareé oscillator.

The forced Poincaré oscillator is a model of the sleep-wake system and we are going to work with data in the format one would expect for sleep-wake studies: a binary signal that indicates whether the "subject" is sleeping. This signal is a nonlinear transformation of the state of the system; much information about the state is lost.

A mathematically interesting thing about the forced Poincaré oscillator is that the components of the system have very simple behavior: the unforced oscillator is periodic. Even when the forcing input is also periodic, the output of the system can be complicated.

In radial coordinates the unforced oscillator is given by a pair of uncoupled differential equations:

$$\frac{dr}{dt} = \alpha r(1-r) \quad (1)$$

$$\frac{d\theta}{dt} = \beta \quad (2)$$

The dynamics of the unforced oscillator are simple: the state cycles around the unit circle, the cycle length is set by the parameter β and the speed of approach to the unit circle (that is, the points where $r = 1$) is set by α.

The differential equations could also be written in rectangular coordinates, with $x = r\cos\theta$ and $y = r\cos\theta$. If the state is visualized as a point in a plane, then the type of forcing we will study involves an input that moves the state point over to the right. That is, the forced oscillator is described by

$$\frac{dx}{dt} = f_x(x,y) + u(t) \quad (3)$$

$$\frac{dy}{dt} = f_y(x,y) \quad (4)$$

where $u(t)$ is the input and f_x and f_y are the functions implicit in Eqs 1 and 2.

In experimental studies we do not have access to the state variables (r, θ) (or, equivalently, (x, y). Instead, we measure a quantity that presumably depends on the state. In this exercise, we will measure a Boolean (0 or 1) variable that indicates wakefulness. Thus, our measured time series will look like a square wave.

We will apply inputs $u(t)$ and observe the output $s(t)$. The objective is to use the $u(t) \rightarrow s(t)$ pairs to make deductions about the system. This should, in principle, be easy, since we know the dynamics of the system. But there are substantial difficulties. First, standard forms of $u(t)$ produce steady state behavior. Near the steady state, important aspects of the dynamics are not evident in the dynamics. Second, the $s(t)$ measurement is not in a form that is directly useful for inferring the dynamics.

The primary software for this exercise is the m-file `sleepcycle` which provides an Euler integration of Eqs 3 and 4. (There is no good mathematical reason to use an Euler integration. In some circumstances, the dynamics of Eqs 3 and 4 and the program `sleepcycle` are quite different. Perhaps it's best to think of Eq. 3 as a model of how the data are generated, rather than being reality.)

Along with `sleepcycle` we will use the program `linearcycle` which implements a linear dynamics version

$$\frac{dr}{dt} = -\alpha r \quad (5)$$

$$\frac{d\theta}{dt} = \beta \quad (6)$$

`Sleepcycle` is written with a time step of 5 minutes, so that there are 288 points in a 24-hour day. The input $u(t)$ should be arranged as a vector with one point for each 5-minute time step. The input is intended to be interpreted as light: 0 means darkness and 1 is a "usual" level of light. Typically, in a Poincaré model of sleep-wake cycle, the duration of the input is much shorter than the length of daylight, perhaps corresponding to an accomodation of the light sensitivity mechanism as the day wears on.

Here is an input that corresponds to 100 days of a 24-hour periodic exposure to light, where the effective duration of the light each day is approximately 4 hours (50 out of 288 points):

```
≫ u = rem(1:(100*288), 288) < 50;
```

To compute the output, one runs `sleepcycle` with `u` as an input:

```
≫ s = sleepcycle(u);
```

In this case, both $u(t)$ and $s(t)$ are square waves.

The program `rasterplot` takes $s(t)$ as input and plots it in the manner used for sleep-wake studies. The software has been written so that it works in coordination with `sleepwake` or `linearcycle`, using a day-length of 288 steps. But, by giving a second argument to `rasterplot`, you can set the day length for analysis to be anything you like.

If you use the parameters shown above to set the input $u(t)$, the raster plot will show that the sleep cycles are lined up from day to day. This is called `entrainment` of the oscillator.

- Make the amplitude of the forcing input $u(t)$ a bit smaller (for example `sleepcycle(0.1*u)`) until entrainment fails.

- Set the amplitude of the forcing input to be just large enough to get entrainment. Now change the period of the forcing input $u(t)$ until entrainment fails. For example, to set the period of the forcing input to 270, you would use

  ```
  ≫ u = rem(1:(100*288), 270) < 50;
  ```

  When making a raster plot of the output, you would want to set the second argument to `rasterplot` to whatever is the period of the input.

From this, you should be able to see that, depending on the parameters of the input, the output may or may not be entrained to the input; The output may or may not have the same period as the input.

Try the analysis again, but using `linearcycle` instead of `sleepwake`. Can you make entrainment fail? Why not?

Finally, experiment with the parameters of the input to try to get complicated-looking aperiodic output from `sleepwake`. Does `linearcycle` ever produce complicated output for a periodic input.

Studing the Dynamics of Finger Motion

We are going to use data that you generate yourself in order to explore some of the techniques we have used. The physiological system will be biomechanical: the trajectory traced by your finger as you drag it around the touch pad. (You can also use a mouse, if you prefer.)

Some basic questions we will try to answer using the methods we will study during this course:

1. Is there evidence for nonlinear dynamics in the finger motion?

2. Can you distinguish between the motion of your left and right fingers? Can you distinguish between different people using their finger motion?

   One issue you may need to deal with is the different velocities of different people. One way to deal with this is to resample the data at a fixed number of samples per cycle. Another possible way is to use the x, y instantaneous coordinate as the state, rather than a lag embedded construction.

3. Are the x and y components of the trajectory coupled? (The answer, of course, is that they are. But if I had given you the x signal as one channel of EEG and the y signal as another channel, it might not be so obvious. We will use this as an exercise to try out various ways to detect coupling.) [Example: show that the phase velocities are very similar. show that y can be predicted from a lag-embedded version of x. Maybe do this even for figure 8s. Do a transfer function analysis: maybe this will work well for circular motion, but will it work for figure-8 motion?]

4. When you simultaneously move your left finger and your right finger (being recorded on different computers), is there evidence for coupling between the two fingers?

   It will be interesting to look for synchronization both when you move the two fingers at the same speed and when you try to move one finger faster than another.

Collecting Data

I have written a special suite of programs for collecting finger motion data within Matlab. To use it, add the `FingerMotion` directory to the Matlab path. Running the program

```
≫ setUpPoints
```

will produce a graphics window. Dragging the cursor (that is, moving the mouse with the left button held down) will cause a trajectory to be drawn within the window. Each time you lift and press the mouse button, a new trajectory will be created and stored along with the previous trajectories.

Make several trajectories with your right finger. You might want to make circles, or figure 8s.

All of the trajectory information will be stored in three global variables. In order to have access to these variables

from the Matlab command line, you need to declare the variables:

```
≫ global x

≫ global y

≫ global when
```

You only need to do this once in each Matlab session.

These variables are sampled at a rate that may not be completely steady: it depends on the particular computer and operating system you are using. It's helpful to reduce the data to and evenly sampled set. The special-purpose program `resampleMotion` will do the job:

```
≫ right = resampleMotion(x,y,when);
```

By default, the sampling frequency will be 20 Hz.

The result, stored in the variable `right`, will be a cell array containing one or more $n \times 2$ array. Each array will consist of the x and y coordinates of the trajectory evenly sampled in time.

Going into the graphics window and pressing the 'q' key will erase the memory so that a new set of trajectories can be collected. Do this, then repeat the data collection using your left hand, storing the resampled data in a variable called `left`. Then, so that you don't have to collect new data, save the `left` and `right` data into a file with your own name:

```
≫ save danny-finger left right
```

In order to collect simultaneous finger movement data from both left and right hands, you will need to set up two computers, each running the data acquisition software. On the left computer, store the output of `resampleMotion` to a file (e.g., `danny-simul-left`) and on the right computer store the output to the correspondingly named file (e.g., `danny-simul-right`). Then use whatever file transfer procedure is appropriate to bring the files into a place where you can work with them both on one machine.

2 Lag Embedding, Poincaré Maps and Sections

Objectives

These exercises demonstrate two important techniques in nonlinear time series analysis: processing data sampled from a continuous-time system to produce a discrete-time representation via the Poincaré section and Poincaré map; and constructing a multi-dimensional representation of a single signal using lag embedding. Some issues to examine are how the choice of cut in a Poincaré section affects the results and how the choice of embedding lag affects the gross shape of the "attractor."

Data Sets

Several synthetic and real-world data sets are provided:

Rossler The full x, y, z state of the Rossler system is sampled by the program `makerossler`. This integrates the differential equations of the Rossler system for whatever duration and whatever initial condition you specify. You can set the time between samples. For example,
`≫ [t,x,y,z] = makerossler(1000, [1 0 0], .1);`
will integrate the Rossler equations starting at an initial condition $x = 1$, $y = 0$, $z = 0$ and sampling every 0.1 second.[1]

Several plots of the x, y, z data are easy to make: A time series plot of one variable:
`≫ plot(t,x)`
The trajectory in the true state space
`≫ plot3(x,y,z)` You can use the rotation tool in the Figure window (the button with an arrow chasing it's own tail in the Figure window): press the button and then click and drag in the Figure window to change the perspective on the three dimensional plot.

To make an animation of the trajectory, use
`≫ comet3(x,y,z)`
You can rotate this perspective using the rotation tool while the animation is in progress.

Lorenz Gives the full x, y, z state of the Lorenz system.
`≫ [t,x,y,z] = makelorenz(100,[2 0 20], .01);`

Normal sinus rhythm The file `nsr.dat` contains an intracellular electrogram from normal sinus rhythm, provided by Nitish Thakor at Johns Hopkins University. You load this time series with the command
`≫ load nsr.dat`
which creates a variable `nsr`. You can plot it as a time series with
`≫ plot(nsr)`
or, say, the first 1000 points with
`≫ plot(nsr(1:1000))`
Since this is a single-dimensional measurement, you can't plot out a "state space" — first, you'll have to construct the space.

Ventricular fibrillation The file `vfib.dat` contains an intracellular electrogram during ventricular fibrillation, again provided by Nitish Thakor. Load the data with the command
`≫ load vfib.dat`
which will create a variable `vfib`.

Measles Month-by-month measurements of the number of measles cases in various cities are in the files `balt.dat`, `newyork.dat`, `detroit.dat`.

[1] "Second" should really read "time unit." There are no physical units specified in the Rossler equations.

And, of course, you have the finger data you collected in the exercises on Linearity and Nonlinearity.

Tools

These exercises are mainly about the visualization of time series and trajectories in real or reconstructed state spaces. In later exercises we will deal with analysis and quantitative characterization of these trajectories.

The fundamental tools for visualization are, of course, plotting tools. We've already seen some of these: `plot`, `plot3`, `comet3`.

Another convenient plotting tool is

scatplot(x,tau) takes a scalar time series and makes a delay plot, a plot of $x(t+\tau)$ vs $x(t)$. By default, τ is 1. For example:
≫ `scatplot(x)`
or
≫ `scatplot(x,20)`

Sections

To create a Poincaré section, the following tools can be used:

psection `[t,p] = psection(traj,`*`crossing level, plane, times`*`)` is used to make a classical cutting-plane Poincaré section. `traj` is the collection of time series describing the trajectory. The returned variables are: `t`, the times at which the plane was cut; `p`, the position in the trajectory space of each pass (in a negative direction) through the plane. It is a matrix where there is one variable in each column. (It is very boring to make a Poincaré section of a single variable! Try it sometime.) The other arguments, shown in *italics*, are optional. The "crossing level" and "plane" describe where the cutting plane is located. By default, the plane cuts the first variable at the mean of that variable. For example, taking `x`, `y`, and `z` to be from the Rossler system, try
≫ `[tt,pp] = psection([x y z]);`
Note that the three signals have been combined into one matrix to be handed off to `psection` and that `psection` returns two variables, the times of the crossings `tt` and the position `pp` in the full space of the crossings. There is one row for each crossing. Since the default cutting plane is the first variable, you'll see that the first column is constant. The other columns contain the information you want.

In this case the space being cut is three-dimensional, and the cut itself is two-dimensional. You can plot the places where the trajectory passes through the cutting plane with
≫ `plot(pp(:,2), pp(:,3), '.')`
(If you had used another cutting plane than the default one, you would have had to plot the points differently.) For example, try
≫ `scatplot(pp(:,2))`

The Poincaré map is the relationship between the position of one pass through the plane with the next pass through the plane. Plotting this out fully would require 4 dimensions. For the Rossler data — but not for all data generally — the passes lie practically on a one-dimensional curve. We can exploit this structure to create a Poincaré map as a two dimensional plot. Try
≫ `scatplot(pp(:,2))`

dmax(data,timescale,thresh) finds local maxima in a scalar time series.[2] The `timescale` sets the meaning of "local." For an oscillatory signal, it should be set to be about somewhat less than the time between peaks — the precise value usually isn't critical. The optional argument `thresh` is used to discard local maxima that are not large enough to be of interest to you. `dmax` returns the times and heights of the maxima detected. By plotting these out along with the original signal, you can adjust `timescale` as appropriate. For example,
≫ `[tt,aa] = dmax(balt,8);`
The time scale has been set to 8, since this is somewhat less than the once-a-year peaks expected in measles.
≫ `plot(balt); hold on; plot(tt,aa,'o');hold off` The plot suggests that some of the small maxima have been missed. You can change the time scale to pick up these peaks if you like.

Embedding

Lag embedding is the fundamental technique in nonlinear time series analysis. This takes a scalar signal $x(t)$ and turns it into a vector signal; at each time t the vector is $(x(t), x(t-\tau), \ldots x(t-(p-1)\tau)$ The parameter p is called the embedding dimension, the parameter τ is the lag.

lagembed(x,p,tau) takes a vector `x` and returns a matrix that is the lag-embedding of embedding dimension `p` and lag `tau`, both of which should be integers.[3] Try
≫ `traj = lagembed(balt,4,2);`
Note that `lagembed` doesn't create any new information; it just re-organizes existing information. Each column of the matrix `traj` is the original signal shifted a bit in time (and with a bit cut off the ends of the signal).

[2] Although `dmax` may not seem to be a Poincaré section, you might want to think of it as a section in the space constructed from the signal and its first derivative: the maximum of the signal corresponds to the plane where the derivative is zero.

[3] In principle, τ does not need to be an integer, but here we are dealing with sampled data.

Finally, a small program that keeps a "sub-space" of a collection of data stored as a matrix:

keepPC(data,nkeep) keeps the `nkeep` largest principal components of the cloud of `data`.

Questions

1. Embed the time series
   ≫ `simple = [1:10]';`
   (making sure it's a column vector) for various embedding lags and dimensions. By looking at the matrix produced by `lagembed` you should be able to see exactly what is going on in lag embedding.

2. Take a chaotic time series from the quadratic map (in the exercises on nonlinearity). Embed this time series in 2 dimensions with a lag of 1. Why do the data form the shape that you see? Now embed the chaotic time series in 3 dimensions. Rotate the plot around to get an ideal of the shape of the set of embedded points. Now try using a lag of 2 or more in 2-dimensional and 3-dimensional embeddings. Why does the shape change so much?

3. Take a non-chaotic, periodic time series from the quadratic map (from the first set of exercises). Embed this in various ways and explain the "shape" of the embedded data.

4. Plot the Lorenz data in the original x, y, z coordinates to see the shape of the attractor in the actual state space. Now use lag embedding on just a single coordinate to produce a reconstructed model of the attractor. Since we're visualizing the attractor, we're pretty much restricted to using a dimension of $p = 2$, so you can use `scatplot` to form and plot the embedding in one command. You might also want to use `lagembed` with a dimension of $p = 3$ along with `plot3`. Note that `plot3` takes three vectors as arguments, so you will have to do something like:
   ≫ `foo = embed(x,3,10);`
   ≫ `plot3(foo(:,1), foo(:,2), foo(:,3));`

   Find an embedding lag that gives the most faithful-looking reconstruction of the trajectory; compare the reconstructed trajectory to the original `[x y z]`. Try lags that are very short, lags that are a fraction of the typical period of oscillation of the signal, and lags that are longer than the period of oscillation. Try embeddings of all three state variables individually.

   The program `acf(data)` computes the autocorrelation function of a scalar signal. One rule of thumb for selecting an embedding lag is to pick the lag that corresponds to the first zero-crossing of the autocorrelation function or, for signals without regular oscillations, the lag at which the autocorrelation function falls to about $\frac{1}{e} \approx 0.37$. (You can use `acf(data,maxlag)` to restrict the plot to lags less than `maxlag`.)

5. Construct a Poincaré section of the Rossler data and generate a Poincaré map from this. How do the results depend on the way in which the Poincaré section is taken, that is, on the position and orientation of the cutting plane?

   Create a Poincaré map of the Lorenz data. It is not as easy as it might seem to create a map that shows a simple structure. It's hard to find a good Poincaré cutting plane. Try both the classical Poincaré cut (`psection`) and local maxima of one signal.

6. Create lag embeddings of the real-world signals, picking the embedding lag to produce a "chaotic-looking" trajectory. Experiment with ways to take Poincaré sections of these reconstructions. For the cardiac data, a typical Poincaré section is measuring the reconstructed model state once per "beat." For the measles data, the section is measuring things once per outbreak.

7. The use of principal components allows one to use a rather high embedding dimension and project this down into a lower-dimensional space while retaining much of the (linear) information in the signal. Try using a very short embedding lag — much shorter than you found in (1) — and a high embedding dimension. The product of the embedding dimension and embedding lag is called the "embedding window" and reflects the amount of the signal that is incorporated in each point in the embedding. Use `keepPC` to keep the two or three largest principal components of the embedding. Plot these out and compare them qualitatively to the results you got using the best embedding lag you found in (1).

3 Dimensions

These exercises are about the correlation integral and correlation dimension. Calculating the correlation integral is straightforward once the embedding parameters have been chosen. The correlation dimension is derived from the correlation integral, but there are a number of ways of doing this; the theoretically correct method is impossible in practice. Interpretation of the correlation integral is therefore somewhat difficult.

We'll look at the correlation dimension for a variety of signals: chaotic, random, and deterministic but with a random component.

Tools

The main program is `c2`, which computes the correlation dimension of a time series. The program lag embeds the time series. The mandatory arguments are:

- the time series;
- the embedding dimension;
- the embedding lag;

As you can see, these arguments have to do with creating an embedding of the time series. If you already have an embedded set of points, then use the program `c2embed`. The first argument to `c2embed` is the matrix of embedded points.

Some additional arguments are optional for both `c2` and `c2embed`.

- the decimation factor. The `c2` program is very slow because it is written in an interpreted computer language. In order to speed things up, the program is arranged to allow you to calculate a sample of the correlation integral. Rather than using every reference point in the time series, a sample of reference points is used. By default the decimation factor is 1, but increasing it will correspondingly decrease the computation time (at the cost of a less precise correlation integral).

  USE A LARGE DECIMATION FACTOR at first, decreasing it when you find the computation time is satisfactory. What's large? If your time series is length N, then the computational time goes as $\frac{N\hat{}2}{\texttt{decim}}$. You should set `decim` so that this number is no larger than, say, 100,000. That is, for a time series of length 1000, set decim to be at least 10 at first. If you get results that you want to refine, you can lower `decim`. For a time series of length 1000, with `decim` set to 10, you can expect the correlation integral calculation to take some tens of seconds. For a time series of length 10000, `decim` should be set to something like 10000, decreasing it as you decide to invest in a more precise calculation.

- the Theiler correction factor. This factor is intended to reduce the influence of linear correlations on computations of short time series. It's default value is reasonable, so you don't need to worry about it except when you want to see what happens without it.

- the embedding dimensions to report. Again, this is worth worrying about only if computation time is an issue.

The program returns two variables:

r a vector of length scales at which the correlation integral has been calculated.

c the correlation integral. Each column is the integral at one of the embedding subspaces. That is, the first column is for an embedding dimension of 1, the second column is for an embedding dimension of 2, and so on. (If the `reportdims` variable is set, then the columns are for the corresponding dimension in `reportdims`.)

It's really much easier to use than the above suggests. Let's try it:

```
≫ load rossler.dat
≫ x = rossler(:,1); % the x component
≫ [r,c] = c2(x,3,3,10); % decimation 10
≫ plot(r,c); legend('1', '2', '3')
```

The `legend` command helps to label the curves: there's one for embedding dimension 1, one for dimension 2, and so on. (Use the mouse to drag the legend box to another place if it's in the way.)

Another program, `c2embed` is exactly the same as `c2` but it takes a matrix instead of a time series. This is useful if you already have an embedded cloud of points, perhaps from principal components of an embedding or some other source.

If we plot the correlation integral on log-log axes, then the slope is the correlation dimension.
`≫ plot(log(r), log(c)); legend('1','2','3');` You can see that the slope is not the same everywhere; this is the source of some difficulty with the correlation.

We need to pick part of the curve to calculate the slope. One easy way to do this is to specify the upper and lower bounds on the vertical axis since often the curves appear quite straight over the same vertical range. The program `d2byC` will do this:

```
≫ dims = d2byC(log(r), log(c), 6, 8)
ans:     ans:  0.9924
         1.7963
         1.8626
```

To illustrate a somewhat longer calculation, we'll calculate the correlation integral for embedding dimensions up to 10:

```
≫ [r,c] = c2(x(1:1000),10,3,10);
≫ plot(log(r), log(c)); legend(num2str([1:10]'));
≫ xlim([-5 5]) % set the x-axis to a better scale
≫ dims = d2byC(log(r), log(c), 5,7)
ans:     ans =
         0.9569
         1.7803
         1.9395
         1.9788
         1.9347
         2.0032
         2.0156
         2.0489
         2.0570
         2.0559
```

Researchers often plot the computed dimension versus the embedding dimension
`≫ plot(1:10, dims, 'x-')`
and look for "saturation," a leveling-off of the curve. Here the curve levels off at a dimension of slightly more than 2.

Questions

- Compute the correlation integral of a very simple set of points embedded in 1 dimension:
  ≫ `x = [1;2;3;4;5;6;7;8;9;10];`
  (Make sure that `x` is a column vector, that is, a matrix with only one column.) Now compute the correlation dimension of these points (in their 1-dimensional embedding):
  ≫ `[r,c] = c2embed(x);`
  ≫ `plot(r,c)`
  Explain the result in detail. (Note, here we've plotted `r` and `c` on linear axes. In the remainder of these exercises, we'll use log-log axes.)

- Compute the correlation dimension of gaussian white noise:
  ≫ `noise = drand(1000,1);`
  ≫ `nn = lagembed(noise, 10, 1);`
  How does the correlation dimension depend on the embedding dimension?

- Compute the correlation dimension of a periodic sine curve:
  ≫ `s = sin([1:1000]/8)';`
  How does this depend on the embedding dimension?

- Take a nonlinear measurement transformation of the sine curve, for instance
  ≫ `x = s.^3;`
  or
  ≫ `x = sin( 3*s );`
  Do the dimensions differ from those of the original sine curve?

- Add a small amount of noise to the sine curve
  ≫ `x = s + 0.1*randn(size(s));`
  How does this change the correlation integral?

- Take a length of the Rossler data generated with `makerossler`. Arrange to sample the signal so that there are roughly 25 points per cycle and about 1000 points altogether. What is the dimension? Now generate another such signal from a different initial condition. Add it to the first signal. What is the dimension of the sum?

The Chaos Game

A simple stochastic mechanism to generate a fractal is called the "chaos game." (It's not chaotic, because it's random, so the name is somewhat misleading.)

In the chaos game, you specify a set of target points, a dilation factor, and an initial position. At each step, you pick one of the target points at random, and move to a new point from the present position. The new point is closer to the target point by the dilation factor. For example, if the dilation factor is 1/3, the new point will be 1/3 as far as the present position. When starting the next step, the new point is taken as the present position.

As an example, consider the target points that are the three corners of an equilateral triangle:
≫ `targs = [0, 0; 1, 0; .5, 0.866];`
We'll set the center of the triangle
≫ `pos0 = [0.5, 0.433];`
Using a dilation factor of 1/100, we take one step:

```
>> seq = chaosgame(targs, .01, 1, pos0)
seq =
    0.5000    0.4330
    0.9950    0.0043
```

The new point is almost all the way to the target point at $(1, 0)$. In the next step of the game, we start at $(0.995, 0.0043)$, pick one of the target points at random, and move toward it.

The `chaosgame` program returns the coordinates of the points visited in the course of playing the game. Try the chaos game with the above target points, and a dilation factor of 0.5. Run 1000 iterations from `pos0` (or any other initial point you like). Then plot out the points. The statements will look like this:

```
>> seq = chaosgame(targs, .5, 1000, pos0);
>> plot(seq(:,1), seq(:,2), '.');
```

The resulting plot should look like the Sierpinski gasket.

The points in the chaos-game sequence (aside from an initial transient), lie on a fractal whose dimension is $-\frac{\ln N}{\ln D}$, where N is the number of target points and D is the dilation factor. For example, the Sierpinski gasket has a dimension of $-\frac{\ln 3}{\ln .5} \approx 1.585$. (The formula for dimension assumes that none of the target point "copies" overlaps with another. This will be true so long as the target points are sufficiently spread out.)

You can play the chaos game in any dimension you like:

- Try the 1-dimensional target points 0 and 1, with a dilation factor of $\frac{1}{3}$. (The points need to be arranged as a column vector, `[0;1]`. To plot the resulting sequence, you need to trick MATLAB into thinking that there are two coordinates, something like `plot(seq, zeros(size(seq)), '.')`)

- Try the 3-dimensional target points that are the corners of the unit hypercube. Use a dilation factor of $\frac{1}{2}$.

The chaos game gives us an easy way to generate data with a known fractal dimension. This allows us to explore how precise the correlation dimension estimate is for any given amount of data. Note that you do not need to embed the sequence that results from the chaos game: it is already effectively embedded. You should use the `c2embed` program since the points in the sequence are already embedded.

1. How much data do you need to distinguish between a dimension of $\frac{\log 3}{\log 3}$ and one of $\frac{\log 3}{\log 4}$? (Note: Three target points, dilation factors of 3 and 4 respectively.)

2. How much data do you need to distinguish between a dimension of $\frac{\log 8}{\log 3}$ and one of $\frac{\log 8}{\log 4}$? (The 8 target points might be the corners of the unit 3-dimensional hypercube.)

3. How much data do you need to distinguish between a dimension of $\frac{\log 16}{\log 3}$ and one of $\frac{\log 16}{\log 4}$? (The 16 target points might be the corners of the unit 4-dimensional hypercube.)

4. Try increasing the dimension, using 1000 sequence points. What is the largest dimension you can reliably find?

Animal EEG Data

Examine some of the animal EEG data. Start by plotting it out, constructing simple 2-dimensional embeddings to see the pattern of points, looking at some Poincaré sections. Do all this before computing a dimension.

Do you see any reason to think that the dimension estimates might be misleading? How might you remove artifacts from the data?

4 Fitting and Predicting with Locally Linear Models

Objectives

These exercises will introduce some concepts from function estimation. The emphasis will be on how to construct dynamical models from data and how to use those models to generate synthesized data, make predictions, and measure the determinism of data.

There are, of course, many different types of models. The exercises here will focus on one simple type: the locally linear model. This is not the best type of model for all purposes, but it is generally very good for most purposes. It has the great advantage of having only one parameter that needs to be set: the number of neighbors to use in fitting the locally linear model. It is easy to find reasonable values of this parameter. Of course, you may also need to embed a time series in order to generate the model, so the embedding parameters are needed as well.

Whatever the technical architecture of a model, the abstract structure is the same. Given an input, the model produces an output that corresponds to the input. There are two basic steps:

Training In the training stage, often called *fitting*, we provide some inputs and corresponding outputs. The model "learns" from this training data how to translate from an input to an output.

Evaluation Using the trained model, we can now provide an input. The model, having already been trained, translates this input into an output.

We will focus on how to set up the training data, and various ways of evaluating the model.

The Training Data

We will be modeling dynamical systems. In a dynamical system, the present state determines the future state. Our data, when suitably processed and embedded, provide a measurement of the present state. We will call these measurements the "pre-image." The system's dynamics translate the pre-image into the future, which we'll call the "post-image."

In order to train the model, we need to provide the pre-images and the corresponding post-images. As a simple example, consider some data from the quadratic map:

≫ `load quad.dat`

Plotting the first few points versus time shows the data look irregular

≫ `plot(quad(1:100)`

but plotting $x(t+1)$ vs $x(t)$

≫ `scatplot(quad))`

shows how the past value, $x(t)$ determines the future value, $x(t+1)$. In this case, the state is a scalar and an appropriate state space is one dimensional.

Although the `quad` data is 1000 points long, let's take just the first 500 points as the training data. First, we need to take the pre-image data:

≫ `pre = quad([1:500]);`

This[4] gives the state of the system at each time 1 to 500; there is one row for each time point.

Next, we need to take the post-images that correspond to each of the 500 pre-images. At each time, t, the post-image is $x(t+1)$. So, our post-images are[5]

≫ `post = quad(1+[1:500]);`

But, you object. The post-images are the same data as the pre-images. This is true. But each row in the post-image is different than the corresponding row in the pre-image. It is the nature of dynamics that the post-image at time t is the pre-image at time $t+1$.

Let's do an example more generally. Suppose we take the x, y, z signals from the Lorenz system. Then the state of the system as it evolves over time can be represented by the matrix

≫ `state = [x y z];`

[4] Of course I could have written the command as ≫ `pre = quad(1:500);` Writing `[1:500]` instead of `1:500` gives exactly the same result. However, later I'll be adding 1 to each of the points in `1:500`.

[5] Following up on the last footnote, note that `1+[1:500]` in the command below has the same meaning as `2:501`. Writing `1+1:500` would be equivalent to `2:500` which is not what we need.

Each row in this matrix gives the state at a single time. Suppose we wish to take as our training data the first 999 points of this state:
≫ `pre = state([1:999],:);`
The corresponding post-image is the same set of states, but offset by one time step[6]
≫ `post = state(1+[1:999], :);`

Similarly, we might have a state reconstructed via an embedding of a measured signal. The embedding is a matrix and we construct the pre- and post-images for the training data in the same way as above.

Do keep in mind that if you have N rows in your state matrix, you will not be able to use all N of them as pre-images. The reason is that you need to have a post-image for each pre-image, and so you could use at most $N-1$ of the rows as pre-images. Things can be even more restrictive, since we will often want to reserve some of our data for evaluation of the model.

Tools

The main program is `linpredict`. This is a very general program. The program takes at least 4 arguments:

pre The training pre-images.

post The training post-images.

k The number of neighbors to use in constructing the local linear model. This is the only parameter of the model, and it is quite easy to find a reasonable value for it. Here are the general principles:

- `k` must be larger than the dimension of the input space.
- `k` must be smaller than the length of the training set.
- `k` The smaller `k` is, the more local the model and the more jagged the model can be. The larger `k` is, the smoother the model.
- `k` If `k` is similar in size to the number of training data points, the model is essentially globally linear.

newpre another set of pre-images; the ones that you want to evaluate the model at.

The first three arguments set up the training of the model. The fourth argument evaluates the trained model at the given data points. This is potentially confusing, so let me emphasize the point: `linpredict` trains and evaluates the model all in one go.

Here's an example using the quadratic data.
≫ `pre = quad([1:500]);`
≫ `post = quad(1+[1:500]);` Since the state space is one dimensional — there's just one column in `pre` — we need $k \geq 2$. We'll take
≫ `k=10;`

The three variables `pre`, `post`, and `k` provide all the information needed to train the model. But we also need to tell `linpredict` where to evaluate the model. We'll going to specify another set of pre-images, and `linpredict` will return to us the model's corresponding post-images. Since the pre-image space is one-dimensional, we can in this case make a graph of post-image vs pre-image. For the data themselves this is
≫ `plot(pre,post,'.')`
which sketches out the shape of the quadratic function. Let's make a similar graph for the model, by asking `linpredict` to evaluate the model at a series of points from $-.5$ to 1.5:
≫ `newpre = [(-.5):.1:1.5];`
(N.B.: `newpre` should have one column for each dimension in the state space. Since in our example the state space has dimension 1, `newpre` has one column. `linpredict` would complain if you made `newpre` a row vector.) Now, we simultaneously train and evaluate the model
≫ `modelvals = linpredict(pre,post,k,newpre);`
The variable `modelvals` now contains the post-images that correpond to each of the points in `newpre`. We can make a plot comparing the data with the model as follows:
≫ `plot(pre,post,'.', newpre, modelvals, 'x');`
Note that the locally linear model follows the data pretty closely, but it also extends the model beyond the domain of the training data. This extrapolation is linear, because for the extrapolation, no matter how far out we go, the "locale" in "locally linear" is the same points at the extreme of the training data.

In general, the pre-images and post-images will not be one-dimensional, and we will not be able to make plots like the above. Instead, we will need to be able to use other techniques to judge whether the fitted function is reasonable and to use the fitted function to study the dynamics of the experimental system we are interested in.

Evaluating the Fitted Function

What are we to do with this fitted function? The function comprises our model of the dynamics of the system; although it isn't in the form of an algebraic expression, it plays the same role in the dynamics:

$$x_{t+1} = f(x_t)$$

the fitted function is the $f(\cdot)$ that we estimate from the dynamics. We can therefore use it to synthesize data, just as we could if we had an algebraic form for $f(\cdot)$.

Here's an example based on data from the Lorenz system:
≫ `[t,x,y,z] = makelorenz(100, [1 0 10], .1);`

[6] Actually, the offset doesn't need to be 1. It could be any integer, positive or negative. We'll explore this later.

```
≫ state = [x y z];
≫ pre = state([1:500],:);
≫ post = state(1+[1:500],:);
≫ k = 50;
```
Now we have all the information for a model.

Let's evaluate the model at $(0, 0, 0)$; we know that the real Lorenz system has a fixed point at the origin, and so the next state should be the same, $(0, 0, 0)$.

≫ `linpredict(pre,post,k,[0 0 0])` ⇒ *ans: 0.0608 0.2234 4.9179*

So, the model isn't exactly right. But perhaps it captures the essential dynamics of Lorenz even if the details aren't exact. One way to find out is to continue the dynamics starting at the new point we just calculated:

≫ `linpredict(pre,post,k,[0.0608 0.2234 4.9179])` ⇒ *ans: 0.3559 0.8088 7.4691*

We could continue on in this way indefinitely, generating a trajectory of the model dynamics. This is, admittedly, quite tedious. To save typing, the function `freerun` does the job for us:

```
≫ traj = freerun(pre,post,k,[0 0 0],1000);
```
The last argument says to run the model for 1000 steps. We can look at the trajectory in the usual way:

```
≫ plot3(traj(:,1), traj(:,2), traj(:,3))
```

You might imagine using free running of the model in the following way: construct the model and free run it from the first point in the data. If the model is right, then the model trajectory should be close to the data's trajectory. However, one serious shortcoming of this approach is that it uses the same data for constructing the model and for evaluating it. This is called "in-sample" testing and tends to overestimate the quality of the model.

One way to evaluate the similarity of the model dynamics to the real dynamics is to divide your data into two sets, a training set and a test set. This is "out-of-sample" testing. Use the training set to create your training pre- and post-images. Then free run the model from the first point in your test data. Ideally, the model should follow the same trajectory as the test data. By comparing the two trajectories, you can get some idea of how good the model is.

Note that "comparing the two trajectories" and "get some idea" are rather vague. One problem for free-running is that if the real system is chaotic, even if the model is almost exactly right the chaotic sensitive dependence on initial conditions will tend to pull the model data away from the real data. For chaotic data, comparing the two trajectories means something like comparing the shapes and locations of the two attractors, and not expecting the model predictions to stay close to reality for long prediction times.

Even for chaotic data, the predictions may be good for short forecasts. This suggests another strategy, make many short forecasts from the points in your testing set. That is, make a short forecast starting from the first point in the testing set and compare it to the real value. Then do this again starting from the second point in the testing set, and then the third, and so on.

Leave-one-out cross-validation is a way to use all of our data for training, without running into the problems of in-sample testing. The idea is to make many models. For each model, we leave out one point. We use that point to test the model. As a result, we can use all of our data for training even while reserving all of the data for testing!

The function `LOOprediction` runs `linpredict` in a way that implements leave-one-out (LOO) cross validation. To illustrate, let's pick up on the Lorenz example:

```
≫ [preds,actual] = LOOprediction(pre,post,50);
```
The returned variable `preds` contains the predictions from each of the models. The variable `actual` contains the corresponding actual value. Insofar as they are similar, the model is good.

Let's plot the first column of `preds` versus the first column of `actual`: if they are similar, the graph should lie on the diagonal line.

```
≫ plot(preds(:,1), actual(:,1), '.');
```

The residuals are the difference between the predictions and the actual values

```
≫ resids = preds(:,1) - actual(:,1);
```
The smaller the residuals, the better the fit of the model. (We take the first column in the above to translate back from the the embedding to a scalar signal.) A standard way to measure the quality of the model is to take the variance of the residuals: `var(resids)`. Using this, you can calculate an r^2 statistic for the model by taking

```
≫ rsq = 1 - var(resids)/var(actual(:,1))
```
You can also do the standard sorts of regression residual diagnostic plots

```
≫ plot(resids)

≫ plot(resids, actual(:,1),'x'
```
Any structure in these plots suggests that the model is not capturing all of the structure in the data, and that you should try a smaller `k`. (However, a smaller `k` may possibly lead to larger residuals, since fewer data points are being averaged together to produce each prediction.)

Although the measurement of the size of residuals in terms of their variance is almost universal in the context of linear regression, we are performing a nonlinear regression and may not have a good reason to think that the residuals are normally distributed. Nonlinear models sometimes produce very good estimates for most points and terrible estimates for some points (such as the ones at the edge of the cloud of points). Some estimates of the size of the residuals that is robust to outliers are the median square value of the residual, and the mean of the logarithm of the absolute value of the residual:

```
≫ median( resid.~2)
```

```
≫ mean( log( abs( resid ) ) )
```

Although I have been using the word "prediction," for purposes such as evaluating the determinism of a model it is not clear that one wants to do predictions. For example, it might be preferable, acknowledging the role of sensitive dependence on initial conditions in chaotic systems, to make "post-dictions": predict the past. We might even prefer to do something even funnier, "dura-diction": use the past and the future to predict the present. There is nothing mathematically odd about this.

To implement pre- or dura-diction, we can use `lagembed` to create a matrix that is the series of columns of the signal delayed by different amounts. Pull off one of these columns to be the "post-image" and use the rest of them as the pre-image (making sure to delete the post-image column).

```
≫ state = lagembed(x,4,3);

≫ dura = state(:,3);
≫ pre = state(:,[1 2 4]);
≫ [preds,actual] = LOOprediction(pre,dura,50);
```

(Note that `linpredict`, and by extension `LOOprediction`, can use a "post-image" that has a different number of columns than the pre-image.)

Questions

- Free-run the Lorenz data with different values of k. Do the results depend strongly on k? When k is very large (a large fraction of the number of data points in the pre-image set), the global dynamics are approximately linear because the "locale" is practically the whole set. How does this global linearity appear in the trajectories generated by the model?

- Create a free-running model of the Baltimore measles data. Try an embedding dimension of $p = 3$, a lag of $\tau = 4$ and $k = 30$. Run the model for 100 steps starting at `[1 1 1]`. Do the results resemble the original data? (Notice the period of the model as opposed to the original data.) What happens if you run the model further forward in time?

- Use leave-one-out cross validation to assess the modelability of the Lorenz or Rossler data. See if you can use the residuals to help you in choosing an optimal `k`.

- Assess the modelability of one of the real-world data sets. Note that for data that are very rapidly sampled, prediction over the short term may be very easy just because the data don't change very much over the long term. You can make long-term predictions by changing the relationship between the `pre` and `post` vectors. For example:
  ```
  ≫ pre = state([1:500],:);
  ≫ post = state(100+[1:500],:);
  ```
  will build a model that makes predictions 100 time steps ahead. (Such a model isn't suitable for free running, but it may be quite suitable for assessing determinism.)

- One idea for detecting nonlinear dynamics in a model is to try modeling for various `k`, from large to small. If the predictions are better at small `k`, that is evidence for nonlinearity. If the predictions are better for `k` very large — similar to the total number of data points — that is evidence against nonlinearity.

Cross-prediction

In cross-prediction, we use a model constructed from one set of data to make predictions about another set of data. Here's is an example of a possible cross-prediction operation, written as a MATLAB function:

```
function res = crosspredict(sigmod, sigdata, dim, lag, hor
one = lagembed(sigmod, dim, lag);
two = lagembed(sigdata,dim, lag);
[pre1, post1] = getimag(one, horizon);
[pre2, post2] = getimag(two, horizon);

kneibs = 10;

predictions = linpredict(pre1, post1, kneibs, pre2);
resids = post2 - predictions;
res = std(resids(:,1)); % just the first column
```

The steps are:

1. Format each signal so that it can be used to construct a model. This generally means lag embedding the signal, and making a set of pre-images and post-images.

2. Set the model parameters. In this case, it is the number of neighbors to use for fitting the locally linear model.

3. Calculate the predictions that the model-data make about the 2nd data set.

4. The program `linpredict` will take the pre-images of the 2nd data set and generate a model post-image for each of them. (This is not free-running of the dynamics, it is a separate prediction for each pre-image.)

5. Compare the model predictions to the actual post-images, characterizing them as appropriate. Here, we're taking the standard deviation. A small number means that the predictions are accurate, a large number means they are inaccurate. The median absolute deviation would be more robust to outliers.

If you have a set of signals, you can use each signal to cross-predict the others. We can interpret the quality of the predictions as a measure of how close each pair of signals is to

one another. The signals that are very close can be considered to be dynamically similar.

Some things to try:

1. Take two different realizations of the Rossler system (starting from different initial conditions). Is their cross prediction close? You will have to have some way to define "close."
2. In the Rossler-to-Rossler cross prediction, what happens as the prediction horizon becomes large?
3. Cross predict a Rossler and a Lorenz signal.
4. Generate data from the quadratic map for two different parameter values, one chaotic and one periodic (say, period 8). Using the chaotic signal as the model, are the cross predictions good? Does it work to use the periodic signal as the model?

Now think about the finger-motion data. We will try to distinguish between the left and right motion by using cross prediction. Our goal is to find out whether the right-finger data are close to one another but far from the left-finger data.

Before starting to construct the model, think about various "trivial" aspects of the signals: their means, their variances, their frequencies. How might you eliminate any influence the mean and standard deviation might have on the results of your cross-prediction?

Tomorrow we will use surrogate data to see if the different frequencies of the signals is responsible for any differences we find in cross-prediction.

5 Making and Using Surrogate Data

Objectives

This lab introduces surrogate data: how to generate it; how to use it; and how to interpret it. Since generating surrogate data is so simple, there will be a strong emphasis on the interpretation and on pitfalls and artifacts. But don't be too discouraged by the difficulties; whatever are the problems with the surrogate data technique, not using it is much worse! Surrogate data should be a standard part of your nonlinear time series analysis toolbox.

Tools

Three surrogate-generating programs are provided here:

Phase-randomized surrogates `fftsurr` produces a realization of phase-randomized surrogate data. Each time the program is called, it produces a new realization of surrogate data (controlled by the computer random number generator seed).

```
≫ surr1 = fftsurr(data);
≫ surr2 = fftsurr(data);
```

Note that `surr1` and `surr2` are different, but their power spectra (as estimated using the amplitude of the fft) are the same as the original time series:

```
≫ psd1 = abs(fft(data));
≫ psd2 = abs(fft(surr1));
```

Plot out one power spectrum versus the other (`plot(psd1,psd2,'.');`) (or `plot(log(psd1), log(psd2), '.')`) and you will get a straight line: the two spectra are identical.

Amplitude-adjusted surrogates `ampsurr` produces "amplitude-adjusted surrogates." In phase-randomized surrogates, the power spectrum of the test data and the surrogates are identical. However, the actual values in the signal are not the same. For example, try

```
≫ plot( sort(data), sort(surr1), '.')
```

or

```
≫ subplot(2,1,1); hist(data);

≫ subplot(2,1,2); hist(surr1);
```

with phase-randomized surrogates.

In amplitude-adjusted surrogates,

```
≫ asurr1 = ampsurr(data);
```

the amplitudes of the surrogate and data are identical; the surrogate is just a shuffled version of the points in the data. The shuffling has been cleverly done, however, so that the power spectrum of the signal and the surrogate are *almost* the same. Try comparing the power spectra of an amplitude-adjusted surrogate.

Polished surrogates (polsurr) produces surrogates that, like amplitude-adjusted surrogates, have the same amplitude distribution as the original data. But the shuffling has been more carefully done than in amplitude-adjusted data so the power spectrum of the surrogate is even closer to that of the original data.

There is also a program for shuffled surrogates, `surr = shuffledsurr(data);`. This is just like amplitude-adjusted surrogates except the power spectrum is white. The only time you would want to use such surrogates is when all you care about is the amplitude of the data, for example when you want to use shuffled residuals to drive a linear model. *Do not use shuffled surrogates for general purposes.*

To use surrogate data as part of a hypothesis test, you need a test statistic. Some examples of test statistics that are widely used are the correlation dimension and nonlinear predictability. The basic idea is to calculate the test statistic for the original data (called the "test data") and then calculate the test statistic for a bunch of realizations of surrogate data. You compare the single value for the test data to the bunch of values for the surrogates, perhaps ultimately computing a p-value.

Two important points to keep in mind when considering a test statistic are:

1. It should generally be a single number: a scalar, not a vector. There are occasions when a vector test statistic makes sense, but they are more complicated than the ones we will consider here.

2. Exactly the same parameters and settings should be used for computing the test statistic for the test data as for the surrogate data. For example, it is *illegitimate* to pick some parameters for the test data (for instance, the scaling region for the correlation dimension calculation) and then apply those parameters to the surrogates.

If the above two restrictions have been honored, then it should be possible to implement the test statistic as a computer function that takes exactly one argument — a time series — and returns a single number.

Here are some test statistics that are implemented as programs: you can construct your own with the basic template

```
function result = myteststat(data)
result = complicatedcalculation(data, ...
        parameter1,parameter2, and so on);
```

This code would go in a file named `myteststat.m` The complicated calculation might be nonlinear prediction, or whatever; you can put in whatever program you like to do the actual calculations. It is a matter of personal intellectual integrity that the parameters haven't been established by looking at the data you are testing.

Here are two test statistics that are somewhat abstract, but have the great virtue of being extremely fast to calculate:

timerev(data,timescale) computes a statistic relating to the time-reversal asymmetry of the data. The parameter `timescale` is an integer describing a time scale of interest; set it to be perhaps one-quarter of the approximate period of oscillations. In order to avoid setting the parameter by looking at the data, you might want to generate a surrogate and base your estimate of the period from the surrogate. Then throw away the surrogate.

crinkle(data) computes a 4th moment of the time series. James Theiler invented this statistic to demonstrate some weaknesses of surrogate data.

To make it easy to do a hypothesis test, we provide a function `surrogatetest` that takes the data and a function implementing a test statistic, and runs many surrogates. `surrogatetest` returns the t-score and a non-parametric p-value (for a left-sided test, so subtract the result from 1 for a right-sided test). It also gives back the test statistic for the test data and the surrogates. An example (using the Baltimore measles data in `balt.dat`):

```
≫ [t,r]=surrogatetest(balt,'crinkle','fftsurr')
ans:     t = 801
         r = 0.9500
```

The result is a strong indication of nonlinearity: the t-statistic is huge (a value of 1.73 would be statistically significant at the 95% level for a one-tailed test); the non-parametric p-value is 0.05 for the right-tailed test.

By default, 19 surrogates are used. (The number of surrogates is set by the 4th argument, if it is given.) If we ran the test again with more surrogates, the non-parametric p-value would be even less — it's 1% for 99 surrogates. Wow! When have you ever seen a result that strong with so little effort!

An important note about programming. Note that in the above, the test statistic was `crinkle(x)`. The variable should always be `x` regardless of the name of the data set. `x` is just a dummy variable. You can also include parameters, but they must be numbers and not variables. For example: `quickde(x,2,3)` but NOT `quickde(x,dim,lag)`.

Questions

- Re-run the analysis of the Baltimore data with amplitude-adjusted surrogates and with polished surrogates. How and why do the results differ?

- Write a program for either nonlinear prediction or for the correlation dimension and apply it to the lorenz data. Remember, your program should take a data set as an input and provide a scalar number as an output. For the dimension calculation, you might want to use the rule-of-five method. For nonlinear prediction, perhaps take the median absolute value of the residual.

- Take a segment of Lorenz or Rossler data and try nonlinear prediction using both short and long prediction horizons. Now try the prediction on some surrogates generated from the data. How do the results differ for different prediction horizons?

- Let's make some data from a linear process with gaussian white noise inputs:
  ≫ `y = randn(500,1);`
  ≫ `x = arma([1.95 -.96], 1, y);`
  Use the ARMA parameters given. You'll see that the result is a smoothly varying oscillation.

  This represents a signal for which surrogate data should give a negative result: the data satisfy the null hypothesis of surrogate data.

  Test the linear data using surrogate data. Since this is a hypothesis test, you should get a result that causes you to reject the null hypothesis every once in a while; for a significance level of 5% you should anticipate a false rejection of the null 5% of the time. Repeat the test many times, using new realization of the noisy input `y` to generate new linear time series `x`. How often do you falsely reject the null?

- Try the same thing, but using `x.^3` or `x.^2` instead of `x`.

- Let's try a slightly different linear system:
  ≫ `x = arma([1.5 -.7], 1, y);`
  This system has a fast-decaying impulse response.

  Rather than using gaussian white noise as the input, lets use some non-gaussian spikes:
  ≫ `y = zeros(500,1); y(10) = 7; y(287) = -3; y(403) = -5;`
  ≫ `x = arma([1.5 -.7], 1, y);`
  Test this system for "nonlinearity" using the crinkle test statistic. What violates the null hypothesis in this case? Is there a difference in the result for amplitude adjusted and for polished surrogates?

- Generate two ARMA signals, each of length 500 points, but with quite different power spectra. (You can judge the spectrum by eye, perhaps one of the signals is very smooth and the other is jagged.) Concatenate the two signals to make a signal from a linear, nonstationary system. Test the nonstationary signal using surrogate data and a crinkle statistic. Are you able to detect the nonstationarity in the data using the surrogate data technique? If not, make new signals with even more different spectra. If so, try shortening the 2nd signal so that the overall signal consists of 500 points of the first signal concatenated with the shorted second signal. How short can the 2nd signal be before you are unable to detect nonstationarity in the combined signal?

- We are going to make surrogate data from the finger-movement time series. Since there is an x- and a y-component to each time series, we will want to make surrogates of both components. Try doing this separately for x and y. Does the trajectory (in the x, y plane) of the surrogate data resemble at all the original trajectory?

  An important technique when generating surrogates from coupled signals is to preserve the time relationships between the signals, even in the surrogates. You can do this by using the same random phases when constructing each surrogate. The statement would look like
  ≫ `xsurr = ampsurr(x,x,342);`
  where 342 is the random seed being used. (The argument `x` is repeated twice due to the way `ampsurr` was written; the program allows you to give as the second argument a signal whose amplitude distribution you want to duplicate. In amplitude adjusted surrogate data, we use the signal's distribution to set the distribution of the output.)

  By using the same random seen for both surrogates, you preserve the phase relationships between the two signals, even while the phases themselves are being randomized.

  (In writing this, I noticed that the `polsurr` program doesn't have a mechanism for setting the seed. I have written another one, but not yet tested or distributed it. You can help me to test it.)

- Go back to the finger movement data on which you carried out cross prediction to distinguish the left and right fingers. We want to see if it is the linear properties of the signal (such as the angular velocity of the finger motion) that enable us to distinguish left from right. To do this, make a surrogate data set corresponding to each finger movement time series. (Preserve the phase relationships when making the surrogates.)

  Are you able to distinguish left from right when using the surrogate data rather than the original data?